Appears in Intl. Symp. on Dependable Systems and Networks (DSN), June 2007

Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor

Christopher LaFrieda Engin İpek José F. Martínez Rajit Manohar

Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA

http://csl.cornell.edu/

Abstract

Aggressive CMOS scaling will make future chip multiprocessors (CMPs) increasingly susceptible to transient faults, hard errors, manufacturing defects, and process variations. Existing fault-tolerant CMP proposals that implement dual modular redundancy (DMR) do so by statically binding pairs of adjacent cores via dedicated communication channels and buffers. This can result in unnecessary power and performance losses in cases where one core is defective (in which case the entire DMR pair must be disabled), or when cores exhibit different frequency/leakage characteristics due to process variations (in which case the pair runs at the speed of the slowest core). Static DMR also hinders power density/thermal management, as DMR pairs running code with similar power/thermal characteristics are necessarily placed next to each other on the die.

We present dynamic core coupling *(DCC), an architectural technique that allows arbitrary CMP cores to verify each other's execution while requiring no static core binding at design time or dedicated communication hardware. Our evaluation shows that the performance overhead of DCC over a CMP without fault tolerance is 3% on SPEC2000 benchmarks, and is within 5% for a set of scalable parallel scientific and data mining applications with up to eight threads (16 processors). Our results also show that DCC has the potential to significantly outperform existing static DMR schemes.*

1 Introduction

Aggressive CMOS scaling has permitted exponential increases in the microprocessor's transistor budget for the last three decades. Earlier processor designs successfully translated such transistor budget increases into performance growth. Nowadays, however, power and complexity have become unsurmountable obstacles to traditional monolithic designs. This has turned chip multiprocessors (CMPs) into the primary mechanism to deliver performance growth, by doubling the number of cores and exploiting increasing levels of thread-level parallelism (TLP) with each new technology generation. Current industry projections indicate that CMPs will scale to many tens or even hundreds of cores by 2015 [4]. Unfortunately, this does not mean that CMPs are free of power, temperature, or even complexity issues. Moreover, other artifacts intrinsic to deep-submicron technologies render these future "many-core" CMPs increasingly susceptible to soft errors [15, 21], manufacturing defects [6], process variations [3], and early lifetime failures [27].

One appealing aspect of CMPs is the inherent redundancy of hardware resources, which can be exploited for error detection and recovery. Current proposals for DMR-based CMPs statically bind core pairs at design time and rely on dedicated cross-core communication [7, 24, 29]. This presents important limitations. For example, when a core fails due to a manufacturing defect or early lifetime failure, the remaining core in its DMR pair can no longer be checked for hard or soft errors. This effectively doubles the number of unavailable cores for fault-tolerant execution. In the presence of process variations, functional DMR pairs consisting of cores with different frequency or leakage characteristics may have to run at the speed of the slower core, leading to additional performance degradations. Hardwired DMR also presents limitations to effective power density/thermal management, as DMR pairs running code with similar power/thermal characteristics are necessarily placed next to each other on the die.

Instead of relying on a set of rigid, statically defined DMR pairs, we would like a CMP to provide the flexibility to allow any core to form a virtual DMR pair with any other core on demand. We would also like to be able to use additional cores to implement other desirable features on demand, such as TMR, or activity migration to spread heat more evenly on the die without compromising fault-tolerant execution. To do this, we propose *dynamic core coupling* (DCC), a processor-level fault-tolerance technique that allows arbitrary CMP cores to verify each other's execution while requiring no dedicated cross-core communication channels or buffers. DCC offers several important advantages. Specifically, DCC:

- Degrades half as fast as mechanisms that rely on static DMR pairs.
- Facilitates the formation of balanced DMR pairs by selectively binding cores that operate at similar speeds.
- Enables low-power fault-tolerant execution by binding low-leakage cores first.
- Supports existing thermal management techniques based on activity migration seamlessly, regardless of functional core count or adjacency.
- Detects and recovers from both hard and soft errors.
- Provides support for on-demand triple modular redundancy (TMR) at no additional cost, using hot spares.
- Greatly simplifies output compression circuitry and lowers compression bandwidth demand by tolerating large checkpointing intervals that can amortize long compression latencies.

In our evaluation, the performance overhead of DCC over a CMP without fault tolerance is less than 3% on SPEC2000 benchmarks, and is within 5% on a set of scalable scientific and data mining applications with eight threads (16 cores).

This paper is organized as follows: Section 2 reviews the challenges created by CMOS scaling in deep sub-micron process technologies, and explores current fault detection and recovery techniques. Section 3 presents our fault tolerant CMP architecture. Section 4 discusses the additional modifications made to the architecture to support parallel applications. Section 5 describes the experimental setup and reports the results. Finally, Section 6 summarizes our conclusions.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **JUN 2007**	2. REPORT TYPE	3. DATES COVERED **00-00-2007 to 00-00-2007**

4. TITLE AND SUBTITLE **Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Cornell University,Department of Computer Science,Ithaca,NY,14853**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Edinburgh, UK, June 25-27, 2007. U.S. Government or Federal Rights License

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**10**	

2 Background and Related Work

2.1 Deep Submicron Challenges

CMOS scaling in deep submicron process technologies will create significant problems for future many-core CMP platforms. In this section, we review some of these challenges and their implications on fault-tolerant CMP design.

Soft Errors The susceptibility of a device to soft errors is inversely proportional to the amount of charge in its nodes [21]. With technology scaling, smaller transistors and lower supply voltages decrease the amount of charge on a node, thereby making devices more sensitive to soft errors [5]. The soft error rate (SER) of combinational logic in a processor is expected to reach $1{,}000$ FIT (failures in 10^9 hours) by 2011 [21]. Since storage structures can be protected relatively easily by parity or error-correcting codes (ECC), combinational logic is expected to become the dominant source of soft errors.

Manufacturing Defects and Process Variations Manufacturing defects are primarily artifacts of fabrication related failure mechanisms (e.g., open or short circuits), or process variations (e.g., excessively leaky cores). During production, burn-in tests that stress parts under extreme voltage and temperature conditions are used to accelerate infant mortality and to expose latent failures. Once identified, defective cores are disabled and parts are classified into bins based on functional core count. Even among the remaining functional cores that pass burn-in tests, frequency and leakage power can vary.

Manufacturing defects are already posing serious challenges to semiconductor manufacturers: IBM recently announced that many of its nine-core Cell microprocessors will ship with only eight functional cores due to defects, and the company is considering whether to ship chips with only seven functional cores [25]. While no industrial data on core-to-core parameter variability or defect rates in CMPs are available for current generation process technologies, both problems are expected to become progressively more significant with CMOS scaling.

Early Lifetime Failures Although burn-in tests are an effective mechanism to expose latent failures and identify defective cores, testing is by no means perfect; electromigration, stress migration, time-dependent dielectric breakdown, and thermal cycling can all lead to intrinsic hard failures [26] after manufacturer burn-in tests. All of these factors worsen as technology scales. The resulting effect of these failures is permanent in the sense that the device is broken and cannot ever be relied upon to produce correct results. The rate of early lifetime failures for 65 nm technology has been estimated to be $7{,}000$ FIT to $15{,}000$ FIT[27]. It is difficult to draw conclusions about the relative rates of soft and hard errors based on their estimates. However, these projections show the importance of designing a system tolerant to both soft and hard errors.

2.2 Fault Tolerance

DCC falls within a class of fault tolerant architectures that use redundant execution. Redundant execution is a technique that runs two independent copies of a thread and intermittently compares their results. This technique has become increasing popular with the recent shift towards more on-chip thread contexts. DCC is most similar to work that combines redundant execution with simultaneous multithreading [28] (SMT) or chip multiprocessor [16] (CMP) architectures. SMT provides additional thread contexts by allowing multiple threads to use a processor's resources simultaneously. CMPs support additional thread contexts by simply integrating more processors on-chip.

AR-SMT [19] was one of the first proposals to use SMT to detect transient faults. As instructions retire in a leading thread, the A-thread, their results are stored in a delay buffer. A trailing thread, the R-thread, re-executes instructions and compares with results in the delay buffer. SRT [17] builds upon this work by addressing memory coherence between the leading and trailing threads in hardware. Specifically, both threads must see the same inputs from the memory system and produce a single output. SRTR [29] extends SRT by adding support for recovery. We limit further discussion of SMT approaches to SRTR.

CRT [14] uses a CMP composed of processors with SMT support. A leading thread on one processor is checked by a trailing thread on another processor by forwarding results through a *dedicated bus*. The advantage over SRTR is better permanent fault coverage as no resources are shared between a leading and trailing thread. CRTR [7] extends CRT by providing recovery from transient faults. Reunion [23] is a CMP architecture that significantly reduces result forwarding bandwidth by compressing results. These signatures are exchanged between statically bound checking pairs via a *dedicated bus*. Previous work can be categorized by the following: i) synchronization, ii) input replication, iii) output comparison, and iv) recovery.

Synchronization. To compare results in redundant execution, either both threads must be synchronized, or a trailing thread must check the results of all committed instructions against a sequence of forwarded results. It is common in commercial systems, such as The Tandem Himalaya [12] and Stratus [20], to use lockstep execution. The IBM G5/G6/z990 [22, 13] uses replicated fetch, decode and execution units running in lockstep. Lockstep means that each processor executes the same instructions in a given cycle. Lockstep is hard to achieve in SMT and CMP because of contention for shared resources. As a result, SRTR and CRTR maintain a slack between leader and trailer threads using simple queues. The leader thread forwards its results to these queues. The trailing thread reads results from these queues as it issues instructions, thereby aligning trailing instruction execution with leading instruction results. Reunion exchanges compressed results between threads at approximately every fifty instructions. The lag between leading and trailing threads is limited by the number of unverified stores that can be buffered. In SRTR and CRTR, stores are buffered in dedicated queues. In Reunion, stores are buffered in a speculative portion of the store buffer. The maximum lag between threads for these two methods is on the order of a hundred instructions.

Input Replication. There is a potential problem if a trailing thread redundantly executes a load instruction. An intervening store, possibly from a separate thread, may have updated that memory address between the time the leading thread executes the load and the trailing thread executes the load. To remedy this, SRTR and CRTR support input replication via a load value queue (LVQ). When the leader thread executes a load instruction, it forwards the result to the trailing thread's LVQ. The trailing thread reads load values from the LVQ rather than going to memory. Reunion, on the other hand, allows both threads to independently execute load instructions. In the case of an intervening store, Reunion rolls back its state to a checkpoint and executes both threads in a single-step mode until the first memory instruction. The authors refer to this as *relaxed input replication*.

Output Comparison. Faults are detected by comparing the state (register values and stores) of each thread in the redundant execution. In SRTR and CRTR, the leading thread forwards results to the trailing thread's register value queue (RVQ) and store buffer (StB). The trailing thread compares its results with the leading thread's results before committing results. In this fashion, the trailing thread's state is always fault-free. SRTR reduces the bandwidth requirements of the RVQ by only checking register results for instructions at the end of a dependence chain. Reunion greatly reduces the overhead of communicating results between cores by using Fingerprinting [24]. Fingerprinting uses a CRC-16 compression circuit to compress all the new state generated each cycle into

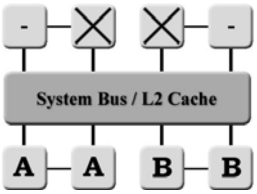

Figure 1. An eight-core statically coupled CMP with two failures. Only two threads can be executed reliably.

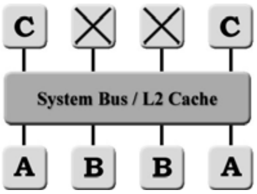

Figure 2. An eight-core dynamically coupled CMP with two failures. Three threads can be executed reliably.

a single 16-bit signature. Comparing these signatures is equivalent to comparing the results of all executed instructions. The probability of undetected error using this technique is very small, roughly 2^{-16}.

Recovery. In SRTR and CRTR, a fault-free state can be recovered by copying the committed state of the trailing thread, which is guaranteed to be fault-free, to the leading thread. This backward-error recovery (BER) technique provides recovery for transient faults, but cannot be used to recover from permanent faults. Reunion checkpoints the architectural state of each core before they exchange fingerprints. When there is a mismatch between fingerprints, the speculative state is squashed and the last checkpoint is restored. The IBM G5/G6 can additionally recover from some permanent faults by copying processor state to a spare processor.

There are two major architectural distinctions between previous work and DCC. First, DCC can recover from permanent faults without the need for constant TMR, like in Tandem and Stratus architectures, or the need for dedicated spares, like in the IBM G5/G6. DCC uses a novel on-demand TMR scheme which only employs TMR during permanent fault recovery. When not recovering from a permanent fault, all processors are configured as DMR pairs and performing computation. Second, DMR pairs in DCC are dynamically assigned. One advantage of dynamic coupling is that a faulty core does not disable both cores of a DMR pair because a working core has the flexibility to form a DMR pair with any other working core. Accommodating dynamic coupling requires architectural extensions over previous approaches. These extensions are described in the following sections.

3 DCC Mechanism

3.1 Architecture Overview

DCC dynamically couples cores by performing all communication between redundant threads over the system bus of a shared memory CMP. In statically coupled CMPs[7, 14, 23], communication between redundant threads is conducted over additional dedicated buses. Dynamic coupling provides the following benefits: i) the system degrades at half the rate of a statically coupled CMP in the presence of permanent faults; ii) when considering variation, cores with similar characteristics can be paired together; and iii) hot spots can be minimized by running high IPC applications redundantly on distant cores. For example, consider the statically coupled CMP in Figure 1 and the dynamically coupled CMP in Figure 2. With two permanent failures, the statically coupled CMP cannot utilize any of the upper four cores for redundant execution because the two working cores can only forward results to the two broken cores. However, the dynamically coupled CMP has the flexibility to pair the two working cores and use them to execute an additional reliable thread. In addition, the dynamically coupled CMP issues the high IPC thread A on distant cores to reduce hot spots.

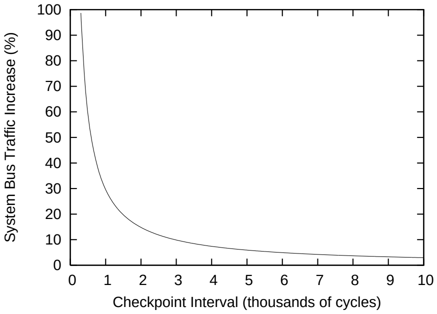

Figure 3. System bus traffic with increasing checkpoint interval size (system described in Section 5).

Utilizing the system bus for redundant thread communication has two major implications. First, communicating over the system bus with a potentially distant core may incur a greater latency than communicating to an adjacent core via a dedicated bus. Second, the resulting increase in system bus traffic could severely impact performance. Figure 3 shows the average increase in system bus traffic over a range of checkpoint intervals (time between output comparisons) for the parallel applications discussed in Section 5. This graph suggests that a long checkpoint interval, roughly greater than $3,000$ cycles, is needed to amortize the increase in system bus traffic. Supporting long checkpoint intervals requires a significant deviation from previous work. For instance, we find that using the relaxed input replication model from Reunion incurs a significant overhead (we evaluate this in Section 5.1.2).

3.2 Private Cache Modifications

In order to support long checkpoint intervals, a large number of memory stores must be buffered. Clearly, thousands of cycles worth of stores will exceed the capacity of the store buffer used in Reunion and CRTR. Instead, DCC's private caches support the cache buffering techniques proposed in Cherry[11]. When a cache line is written, it is marked as *unverified*. Unverified lines are not allowed to leave the private cache hierarchy. Once the buffered state is known to be fault-free, at the end of a checkpoint interval, all unverified marks are gang-cleared. A write to a verified dirty line forces that line to be written back to lower levels of the memory hierarchy, so that it may be restored if a fault occurred during the checkpoint interval. Cherry has shown that this style of cache buffering can easily support thousands of loads with very little overhead (roughly one bit per cache block). In addition to cache buffering support in the private caches, all caches are protected by error correcting codes (ECC), as is done in previous work. It is necessary for each processor to redundantly load data from shared memory into their private cache. However, only one processor needs to write dirty cache lines back into shared memory. We assign the task of writing back dirty data to one processor, the *master*, while the other processor, the *slave*, may evict updated (but verified) cache lines without writing back. The master and slave processors need not be leading and trailing, respectively. Master cores ignore coherence actions to unverified lines by their own slave(s). Conversely, slaves ignore invalidation requests from their own master. Data consistency when running parallel applications is discussed later (Section 4).

There exists a danger of deadlock if an application's unverified dirty lines are allowed to remain in the CMP cache subsystem after the application is descheduled by the operating system. Specifically, the next application to run on the same core may find all cache blocks of a set locked by unverified writes from the previous application, preventing it from making forward progress. If the new application has also locked all cache blocks in a set used by the old application, then a circular dependence arises between the two applications and deadlock ensues. Similar interactions are possible between writes from an application and the operating system. To avoid these problems, we implement a simple policy: be-

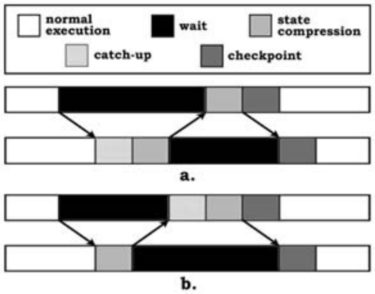

Figure 4. Processor synchronization when: a.) synchronizing processor is leading, or b.) synchronizing processor is trailing.

fore control is transferred across applications and the operating system, all unverified dirty data are verified by scheduling a checkpoint. Once control is transferred, the operating system saves the (just verified) architectural state to the application's process control block as usual.

On a context switch, slave caches are flushed to avoid having multiple copies of a cache line with inconsistent states. If a previous application had used the slave core as its master processor, there is a danger that this earlier application's verified dirty lines may be lost during the cache flush. To avoid such cases, the operating system partitions the cores into master and slave pools, and allocates master and slave cores accordingly. In rare cases where a processor from the master pool may need to be moved to the slave pool, all verified dirty data in the master's cache are first written back to main memory. In cases where a processor from the slave pool needs to be relocated to the master pool, the slave cache is flushed.

3.3 Synchronization

Figure 4 shows our multi-phase synchronization protocol. Synchronization begins when a processor receives a scheduled or unscheduled checkpoint request. Unscheduled checkpoints occur for the following events: i) cache buffering overflow; ii) interrupt; iii) uncached load/store (I/O); or iv) context switch. The processor that receives the checkpoint request sends the number of instructions committed since the last checkpoint to the other redundant processor. The processor initiating the synchronization is either the leading processor, Figure 4a, or the trailing processor, Figure 4b. If the synchronizing processor is leading, the other processor commits enough instructions to synchronize, then compresses and broadcasts its state (via the shared system bus). If the synchronizing processor is trailing, the other processor broadcasts the number of instructions it has committed since the last checkpoint. The synchronizing processor executes enough instructions to match the leading processor, then compresses and broadcasts its state. Once each processor has received the compressed state of the other, it compares against its own. If the compressed states match, a checkpoint is taken and execution resumes. If they disagree, the last checkpoint is restored and the checkpoint interval repeats. In cases where the trailing processor cannot execute enough instructions to match the leading processor (e.g., due to a cache buffering overflow), the last checkpoint is restored, and a new checkpoint is scheduled with half the duration of the last interval. Similarly, if a checkpoint interval does not complete within a fixed timeout period, a rollback is forced and a new checkpoint with half the duration of the last checkpointing interval is scheduled. This eventually guarantees forward progress in cases where a fault prevents the cores from reaching the next checkpoint. Checkpoints are kept in a small, ECC-protected, on-chip SRAM array.

3.4 State Compression

State compression is needed to reduce the bandwidth requirements of comparing state between two cores. Fingerprinting[24] proposed the use of a CRC-16 compression circuit to compress all of the register file and memory updates each cycle. We simulated various parallel CRC circuits[2] in HSPICE and their fan-out-of-four (FO4) delays[1] and transistors counts are shown in Table 1. Assuming a cycle time of 10-15 FO4s, a CRC-32 circuit and a CRC-16 circuit (2 stages) can compress up to 32 bits in one cycle. With potentially more than 256 bits of new state each cycle, a CRC-16 circuit could not keep pace with the core. To remedy this, Reunion uses a multicycle compression scheme. However, the long checkpoint interval in DCC allows us to employ a simple solution that provides a large reduction in required compression bandwidth.

CRC Circuit	Input Width	FO4 Delay	Transistor Count
CRC-16	16	6.65	754
CRC-SDLC-16	16	6.10	888
CRC-32	16	7.28	2260
CRC-32	32	8.60	4240

Table 1. FO4 delay and transistor count for various CRC circuits.

We make the observation that checking the state of the register file at the end of a checkpoint interval is equivalent to checking all the updates made to the register file during a checkpoint interval. Therefore, rather than compressing all the updates to the register file, we simply compress the state of the register file at the end of a checkpoint interval. The time it takes to read out the contents of the 32 entry architectural register file is easily amortized over the long checkpoint interval. In this manner, only memory stores need to be compressed on the fly during a checkpoint interval. Two CRC-32 circuits are needed to compress the data and the address of a store. At the end of a checkpoint interval, these CRC circuits simultaneously compress the integer register file and the floating point register file. For a checkpoint interval of $10{,}000$ cycles, this technique reduces the total state compression bandwidth (i.e., number of bits compressed per cycle) by a factor of 5.4 for SPEC2000 benchmarks.

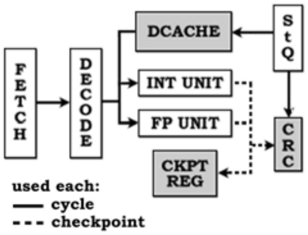

Figure 5. State compression: Stores are compressed each cycle using two CRC-32 circuits, but register values are compressed at checkpoints. StQ is the store queue.

3.5 Recovery

DCC aims to mitigate the impact of deep submicron challenges (Section 2) by endowing arbitrary CMP cores with the ability to verify each other's execution. Detection, however, is only part of the solution; a complete framework for flexible fault-tolerance in CMPs also requires the ability to recover from faults once they are detected. Once again, DCC supports recovery from both hard- and

[1] FO4 is the delay of one inverter driving four identical copies of itself. Delays expressed in units of FO4 are technology-independent.

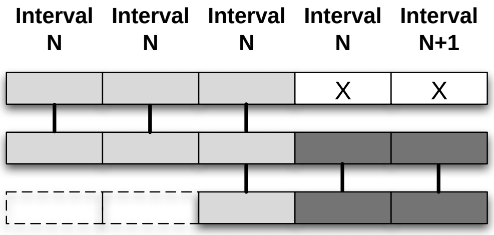

Figure 6. Recovery from a permanent fault using FER after BER fails. After interval N fails twice, interval N is re-executed a third time on three processors. Voting identifies the faulty processor, the system rolls back to the beginning of interval N, and execution continues on the remaining two processors.

soft-errors without requiring dedicated communication hardware or statically binding cores.

When an application requiring redundant execution is switched-in by the operating system, it is appropriated two processors, one master and one slave. Checkpointed register values are stored in the application's process control block by the operating system upon context switches, and are recovered when the application is switched-in. The allocated processors execute instructions, using the aforementioned detection scheme, until a fault is detected. To recover from this fault, backward error recovery (BER) is employed. Both processors rollback architectural state to their last valid checkpoint, invalidate all the cache lines marked as unverified, and resume execution from the checkpoint. If the fault was transient, the processors will successfully complete their next checkpoint. However, if the fault is permanent, the same checkpoint interval will repeat and the system defaults to forward-error recovery.

Specifically, when BER fails to recover from a fault, after repeating the same checkpoint interval multiple times, a third processor is appropriated by the operating system for forward error recovery (FER).[2] (If all other processors are in use, the operating system must choose a core and switch-out its currently running thread.) To initiate this, the cache controller of the failing master core makes a TMR request by generating a special bus transaction that sets a flag in the kernel's address space. This transaction is observed by all other nodes, and a predetermined node is given the responsibility of calling the operating system by jumping to an interrupt vector (each node is responsible for handling another node's requests, and this assignment is made by the kernel). Prior to taking the interrupt, the master and the slave of this remote node synchronize, and then control is transferred to the operating system by jumping to the TMR interrupt handler. The operating system inspects the flags set by the failing node to identify the requesting pair, and allocates a third core for FER. Cases where the requesting node is executing OS code are handled identically. The architectural state of the last valid checkpoint is copied from the master processor to the new slave processor. These three processors, one master and two slaves, implement FER by executing the checkpoint interval in parallel and voting on the correct results (signatures), as shown in Figure 6. Essentially, this amounts to on-demand triple modular redundancy (TMR). Through TMR, the faulty processor is isolated and marked as such, the system rolls back to the beginning of the faulty interval, and all unverified data are invalidated. If the master processor is faulty, all dirty verified data in the master's cache is written back to memory, and one of the slaves is promoted to master. Once the faulty processor is isolated, execution continues on the remaining two cores.

[2] The involvement of the operating system in FER is unlikely to affect system performance since FER is only invoked on hard faults (a rare event) and in cases where the same checkpoint interval fails multiple times in a row due to soft errors (exceedingly unlikely).

State	Cache-line's State Description	Can be unverified?
Invalid	Invalid data	No
Shared	Valid data, possibly inconsistent with memory	Yes
Exclusive	Valid data, consistent with memory, present only in one cache	No
Owned	Valid, dirty data, possibly shared	Yes
Modified	Valid, dirty data, present only in one cache	Yes

Table 2. Summary of MOESI protocol states and whether they can hold unverified data.

4 Parallel Application Support

Our discussion of DCC thus far has been limited to fault detection and recovery for one single node. Although this is sufficient for sequential applications, supporting flexible fault-tolerant execution for parallel applications is at least equally important for future CMP platforms. In this section, we provide extensions that allow DCC to operate correctly when running shared-memory parallel programs. In the following discussion, we define a particular master-slave pair as a *node*, and we refer to all other nodes as *remote nodes*. Recall that master and slave cores that form a node need not be adjacent or even close to one another.

4.1 Checkpoints

Checkpoints are taken globally across all nodes: the bus controller initiates scheduled checkpoints by sending synchronization requests to all nodes at the end of each checkpoint interval. All master threads synchronize with their slaves (as in the sequential case), and send an acknowledgment to the bus controller when the synchronization is complete. When all nodes are synchronized and no outstanding bus transactions remain, the bus controller issues a checkpoint request, and the architectural state on each core is saved. Descheduled threads of a parallel application need not participate in the global checkpoint since their last checkpoint (taken at the time they are descheduled) is still valid. In the case of a signature discrepancy or timeout (Section 3), all processors involved in the execution of the parallel application roll back to their respective checkpoint.

4.2 Coherence

Data sharing in DCC-equipped CMP architectures is different from conventional architectures in that: (1) some of the data held in caches may be unverified–that is, subject to rollback; and (2) sharing decisions must consider whether the processors involved are playing the role of master or slave.

To support sharing of unverified data, we leverage some of the mechanisms previously proposed in the context of the multiprocessor version of Cherry, Cherry-MP [9]. Similarly to Cherry-MP, a natural choice for a baseline cache coherence protocol on which to build DCC support is a MOESI protocol [1]. MOESI allows several copies of a cache line across processors that are possibly incoherent with the copy in memory. Among those copies, the *owned* copy is responsible for (1) providing a copy to any new sharer, and (2) writing back the copy if it replaces the cache line. The other copies remain in the *shared* state. Because DCC requires keeping unverified data off memory, MOESI is convenient for safely sharing unverified modified data across processors. Table 2 compiles MOESI's states; the rightmost column indicates whether the state is apt to hold unverified data. Notice that, in the case of shared state, it is only possible to hold unverified data if there is an owned copy elsewhere in the system–otherwise, the data must be necessarily consistent with memory.

To support sharing of unverified data, we extend the coherence protocol along the lines of Cherry-MP [9] (the Cherry-MP extensions are slightly more elaborate than in DCC because of some additional restrictions specific to Cherry-MP). Specifically:

- Writes always mark the writer's cache line as unverified, similar to the uniprocessor case (Section 3.2). Writes to verified dirty cache lines (modified or owned state) force a writeback of the original contents to main memory, in case a rollback later undoes the update. These writes may be initiated by a local processor, or by a remote processor through a read-exclusive or upgrade request. On the other hand, writes to unverified dirty cache lines must not generate write backs to main memory. If the cache line is marked unverified and dirty elsewhere, the protocol simply forwards the cache line to the requester, if needed.
- On cache-to-cache transfers, the reader's cache line is marked unverified if the original copy of the cache line was marked unverified. This is so that, in the event of a rollback, all live copies of the speculatively updated value are properly discarded. To support this, on a miss, the supplier of the value must put on the bus its cache line's unverified bit as part of its snoop response. Notice that unverified, clean cache lines can be silently dropped by the reader at any time.

To support multiple master-slave nodes, a few more changes beyond those to support a single master-slave pair (Section 3.2) are needed as follows:

- Slaves do not supply data to remote nodes via cache-to-cache transfers–masters do.
- Slave reads cause remote data in modified or exclusive states to downgrade to owned or shared states, respectively. Furthermore, slave read-exclusive requests are treated as ordinary reads for the purposes of state transitions at remote nodes (specifically, they do not result in invalidations). Finally, upgrade requests by slaves are ignored by remote caches. To ensure that remote nodes do not apply invalidations to their caches in response to slave threads, a *master line* is added to the system bus. Cache lines in remote nodes apply invalidations to their caches in response to bus transactions generated by masters only. To facilitate this, master threads drive the master line when they start their bus transactions, and remote threads snoop this line to determine if the bus transaction is generated by a master.
- Read-exclusive requests by a master invalidate remote copies as usual. However, if the data is dirty unverified on a remote master, the slave must obtain a copy of the cache line before it is invalidated by the master. To do this, the slave checks if it already has a copy of the cache line. If so, the cache line is marked as dirty unverified to prevent its eviction. Otherwise, the slave snarfs the cache-to-cache transfer and writes the data to its local cache hierarchy as dirty unverified. If this results in a cache buffering overflow, all cores rollback to the last checkpoint, and a new checkpoint is scheduled with half the duration of the last checkpointing interval (Section 3.3).
- Likewise, upgrade requests by a master invalidate remote copies as usual. However, if the data is dirty unverified on a remote master, *and* if the local slave does not have a copy of the cache line, the master's upgrade request is turned into a read-exclusive request, so that the slave can snarf the remote copy as before. If the slave already has a copy, the cache line is marked as dirty unverified in the slave cache.

4.3 Master-Slave Consistency

The main difficulty in supporting DCC in a parallel execution is ensuring that master and slave threads view a consistent image of the shared address space at all times. In other words, every committed load instruction on the slave should read the same value as the corresponding committed load on the master. In a sequential application, this is easily accomplished by preventing unverified dirty lines from being written back to main memory. Consequently, slave threads can always obtain the same value as their masters from the memory system (Section 3.2).

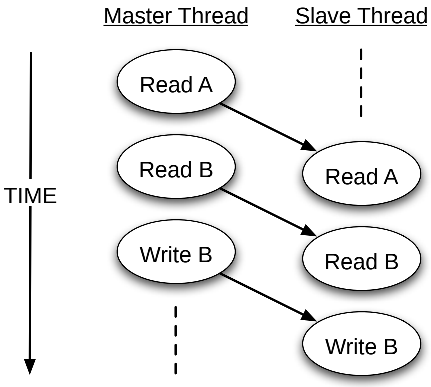

Figure 7. Three example windows.

In a parallel execution, however, this is not necessarily the case. Since all threads in the system have access to the shared address space, intervening writes from other threads can cause master and slave threads to read different values. For instance, the master thread could read a line L, but before the slave gets a chance to perform the corresponding read, a third core could invalidate the master and update L. In this case, the original value that the master read may no longer be available to the slave.

In order to accommodate thread interactions like these, we introduce the notion of a *master-slave memory access window*, or simply *window*. Roughly speaking, windows represent periods of vulnerability during which the consistency of master-slave pairs in the system may be compromised, and allow us to define a small set of restrictions that guarantee correctness in such cases. Windows are defined on a per-address basis, and are labeled as read or write windows depending on whether the operation being performed is a load or a store.

A read window for address A *opens* when *either* the master or the slave thread issues a load that reads the value of A, and *closes* when *both* master and slave threads commit the load. Windows opened by misspeculated loads (e.g., in the shadow of a mispredicted branch) are closed at recovery. Similarly, a write window for address A *opens* when *either* the master or the slave thread performs (necessarily at commit) a store that writes A, and *closes* when *both* master and slave threads commit the store.

Figure 7 shows three example windows on a node. When the master issues its first read, a read window for memory location A opens. The master then opens a read window for B, and commits the load that reads A. When the slave commits its read of A, the open read window for A closes.

To guarantee master-slave consistency, it is sufficient to ensure that the system observes certain restrictions at all times for each memory location M in the program's address space. Specifically, a node should not be allowed to open a write window for M if there is already an open read or write window for M on another node. This *remote intervention constraint* prevents master and slave threads from reading different values due to intervening writes by other nodes. Open windows for different locations place no restrictions on each other. Similarly, any number of simultaneous read windows can be open at a time for a given address, and private data can be read and written without any restrictions. Furthermore, a node can open a read window while a write window is open at some other node.

4.3.1 Hardware Implementation

The main addition that is required to support master-slave consistency is an *age table* that resides with each core's cache controller. Each load's *instruction age* is defined as the the total number of load and store instructions committed by its thread (since the last checkpoint) at the time that load commits. Similarly, each store's instruction age is the total number of loads and stores committed

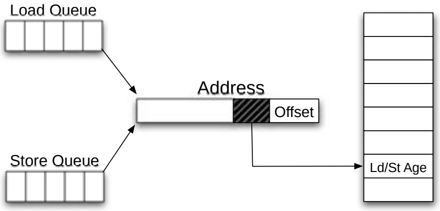

Figure 8. Example of an age table implementation.

by its thread at the time the store commits. We use this information to detect memory operations that could lead to violations of the remote intervention constraint, and to delay them until this danger disappears.

Enforcing the remote intervention constraint requires detecting open read and write windows, and the age table facilitates tracking these by storing the age of the last committed memory instruction on the corresponding core to an address range. The age table is a direct-mapped, untagged SRAM array indexed by the address of committed loads and stores (Figure 8). Age table entries contain the age of the last committed memory instruction to an address range. The table index is formed by using the lower-order adress bits following the cache block offset. The table is updated locally at commit time.

Performing Writes to Modified Data When a master thread wants to perform a write, it checks the state of the line in its cache in parallel with an age table access. If the cache line is modified, no other node is currently caching that line, and there is no danger of invalidating data that the slave will want to read later (recall that unverified dirty lines cannot be replaced until the next checkpoint). In addition, there is no danger of intervention in a read window on another node (if a window had opened following the write that put the line in modified state, the line's state would have transitioned to owned). In this case, the master performs the write immediately.

Performing Writes to Data in Other States If the cache line is not in the modified state, additional checks need to be performed along with the standard read exclusive or upgrade request. The information needed to perform these checks is piggybacked on the bus transaction. When the read-exclusive (or upgrade) request is observed by other nodes, each core accesses its age table to get the age of the last committed memory instruction to the corresponding address range. In parallel, each node searches its load queue[3] for any matching reads that have already issued to the memory system or have forwarded from a store. If there is a hit in the load queue, a negative acknowledgment (NACK) is raised (accomplished by pulling the NACK line of the bus high), and the write is retried later. Searching the load queue guarantees that any read windows opened by speculative reads are not violated by writes from other nodes. In the case of branch mispredictions, speculative loads are naturally removed from the load queue, and misspeculated loads eventually cease to generate NACKs.

If all of load queue searches result in misses, each node reports its age table entry (accessed in parallel with the load queue) on the data bus in the following cycle. (Each slave core drives a portion of the data bus with its age information.) In the following cycle, every master core compares its own age with the age of its slave. A mismatch on a remote node indicates a potentially open read or write window, and a NACK is raised for the write.

Livelock Avoidance There exists a danger of livelock when issuing NACKs for writes. For instance, consider the case of a spin lock where the master holding the lock needs to perform an invalidation for lock release. If masters and slaves on other nodes repeatedly read the lock variable in a tight loop, new read windows may always open before the last one closes, preventing the lock release from performing, and thus perpetuating the cycle. We have empirically observed that NACKs, and thus such livelocks, are exceedingly rare in the applicationswe have studied; nevertheless, we need to provide a machanism to detect these situations and be able to guarantee progress.

We propose a simple policy: A single NACK bit is added to every age table entry; when a node issues a NACK for a write transaction, both master and slave cores set the NACK bit for their corresponding age table entries. At that time, both the master and the slave temporarily stop fetching instructions, and allow their pipelines to drain. When both pipes are drained, the master and the slave exchange their committed instruction counts to identify the leader and the trailer in the execution. The leader remains stalled, while the trailer is allowed to resume execution. This effectively allows the trailer thread to close read windows left open by the leader–in particular, the read windows opened by the spinlock. The remote write (which the remote node keeps retrying) may eventually succeed once the open read window is closed. When this happens, the NACK bit is cleared, and both the master and the slave resume normal execution. It is also possible that the trailer commits enough instructions to match the leader. If at that time the NACK bit has not yet been cleared, the trailer flushes its pipeline and stalls as well. At this point, neither the master nor the slave have any loads in their load queues, and their age table entries are consistent. This guarantees that the very next retry of the remote write will succeed, at which point both master and slave can resume execution.

Deadlock Avoidance There exists also a danger of deadlock when issuing NACKs for writes. This may occur when writes in two or more processors cannot perform because of open read windows by out-of-order loads elsewhere, forming a cycle. For instance, consider the case where processors $p1$ and $p2$ are trying to perform writes to addresses A and B, respectively. If $p1$ and $p2$ have issued loads to addresses B and A out of order with respect to those writes, respectively, the processors will issue NACKs for each other's writes, preveting forward progress. Luckily, this deadlock situation would be eventually broken through the timeout mechanism (Section 3.3). Nevertheless, a simple and more effective solution is possible: Upon receiving a NACK for such a write, a processor systematically flushes its pipeline beyond the write, and prevents loads from issuing until the write successfully commits. In any case, as stated before, we have empirically observed that such NACKs are rare in the applications we have studied.

Hard Fault Recovery Once a signature discrepancy or a timeout occurs (Section 3.3), all processors currently running the parallel application roll back to their last checkpoint. To recover from permanent faults, a second slave is introduced to the node that initially caused the fault. This third processor engages in age exchanges just like the original slave does. When performing an age check, the master thread compares its age against both slaves, and considers a read window to be open if any of the two slaves has a mismatch with it. Aside from this, the operation of our proposed system is the same in all other respects.

4.4 Compatibility Across Memory Consistency Models

The memory consistency model of a multiprocessor places a set of ordering restrictions between memory operations issued from a given thread. DCC does not impose or rely on any ordering constraints, and is general enough to operate correctly under any consistency model. In other words, regardless of how memory operations are ordered, DCC's master-slave consistency is never compromised.

[3] A port already exists for external invalidations to search the load queue in many commercial systems. If this capability is not present, the search can compete with local stores for access to the load queue.

Processor	
Frequency	3.2 GHz
Fetch/issue/commit width	4/4/6
Inst. window [(Int+Mem)/FP]	64/48
Reorder buffer entries	192
Int/FP registers	96/96
Functional Units	4 ALU, 3 FPU, 2 BR, 2/2 Ld/St
Ld/St queue entries	24/24
Branch penalty (cycles)	10(min.)
Store forward delay (cycles)	3
Branch predictor	16K-entry
Branch target buffer size	2048
RAS entries	24
Memory Subsystem	
L1 Cache (Private)	32KB, 4-way, LRU, 64B, 3 cycles
Victim Cache (L1)	8 entries
L2 Cache (Shared)	8MB, 8-way, LRU, 64B, 43 cycles
MSHR entries	16 L1, 16 L2
System bus	256 bits, 800 MHz
Max. outstanding bus requests	96
Memory bus bandwidth	12.8 GB/s
Memory latency	400 cycles
Coherence Protocol	MOESI
Consistency Model	Release Consistency
Fault Tolerance Extension Parameters	
Processor comm. latency	30 cycles
Age table size	64 entries
State compression latency	35 cycles
State checkpoint latency	8 cycles

Table 3. Summary of modeled architecture.

To see this, note that loads obtain their values either from the CMP memory subsystem, or from the store queue of the processor on which they issue. The remote intervention constraint prevents violations of master-slave consistency through the memory system as discussed above. However, relaxed consistency models and aggressive implementations of sequential consistency also allow loads to be reordered with respect to other memory instructions by issuing early, or by forwarding from the store queue. Luckily, DCC readily accommodates such optimizations.

Specifically, if the master forwards from its store queue, a read window opens and prevents intervening writes to violate the window; eventually, the slave also consumes the same value from a replica of the same store (either through its own store queue or from its local cache). In other cases, there is a danger that the master reads a value produced by a remote node, but the slave forwards from its store queue and breaks master-slave consistency. Luckily, the remote intervention constraint prevents this: when the master commits its copy of the store that the slave forwards from, a write window opens and blocks intervening writes from remote nodes, forcing both the master and the slave to consume the result produced by their local store. Windows opened by slaves are also safe for the same reason.

5 Evaluation

Flexible DMR frameworks like DCC hold significant potential when confronted with the challenges of deep submicron process technologies. In this section, we evaluate DCC using detailed execution-driven simulations of a CMP model.

In our experiments, we allow a single application to run redundantly on multiple processors using the hardware modifications described in this paper. The configuration details of this processor are listed in Table 3. When taking global checkpoints on parallel applications, we model the bus arbiter's synchronization request, master-slave synchronization, and the checkpoint latency on each node. We find that master and slave cores are never separated by more than 200 cycles in their execution (roughly 100 cycles on average), leading to negligible waiting times for the receipt of acknowledgments. Hence, for simplicity in our simulations, we do not model the global handshake.

Splash-2	Description	Problem size
BARNES	Evolution of galaxies	16k part.
FMM	N-body problem	16k part.
RAYTRACE	3D ray tracing	car
Spec OpenMP		
SWIM-OMP	Shallow water model	MinneSpec-Large
EQUAKE-OMP	Earthquake model	MinneSpec-Large
Data Mining		
BSOM	Self-organizing map	2,048 rec., 100 epochs
BLAST	Protein matching	12.3k sequ.
KMEANS	K-means clustering	18k pts., 18 attr.
SCALPARC	Decision Tree	125k pts., 32 attr.

Table 4. Simulated parallel applications and input sizes.

To evaluate sequential applications, we simulate 19 of the 26 SPEC2000 suite of benchmarks [8] using the SESC simulator [18].[4] We use the largest datasets from the MinneSPEC [10] reduced input set and run them to completion. To evaluate parallel applications, we use a set of scalable scientific and data mining applications, shown in Table 4. These parallel benchmarks are simulated for 1, 2, 4, and 8 threads, on 2, 4, 8, and 16 processors respectively.

5.1 Results

5.1.1 DCC Overhead

In this section, we assess the performance overhead of DCC over a baseline CMP with no fault-tolerance. We are not concerned with the inherent overhead of executing a thread redundantly on two processors. It is obvious that redundant execution occupies twice the number of cores, and therefore cuts the effective number of processing elements in half. We are interested in the additional overheads involved with orchestrating the detection and recovery schemes presented in this work.

Sequential Applications

We evaluate the performance overhead during fault-free execution by simulating the SPEC2000 benchmarks on a single core in our baseline CMP. We compare this to running the benchmarks redundantly on two cores with checkpoint intervals of 1,000, 5,000, and 10,000 cycles. The slowdown with respect to a single-core execution with no fault tolerance is shown in Figure 9. The average overheads for intervals of 1,000, 5,000, and 10,000 cycles are 20%, 5%, and 3%, respectively. In DCC, a checkpoint takes on the order of 100-200 cycles to complete. Most of this time is spent synchronizing both cores, compressing the register file state and communicating results over the system bus. A checkpoint interval of 1,000 cycles is insufficient to amortize the cost associated with taking a checkpoint. However, an interval of 10,000 cycles reduces this overhead to 3%. One issue that needs to be considered with long checkpoints is their interaction with I/O requests. Prior work [24] has found that checkpoints should be taken at least every 50,000 instructions in I/O intensive workloads to achieve high performance. The applications we have studied obtain an IPC of about 1 on the baseline system, so approximately 10,000 instructions execute in given checkpoint interval, which is well below this 50,000 instruction limit.

Parallel Applications

Parallel applications incur additional performance overheads due to the management of shared variables consistently across nodes (as discussed in Section 4). To see how these overheads scale, we compare speedups under DCC to our baseline CMP for 1, 2, 4, and 8 threads (2, 4, 8, and 16 processors). Table 5 reports the

[4] our simulation infrastructure currently does not support the other SPEC benchmarks

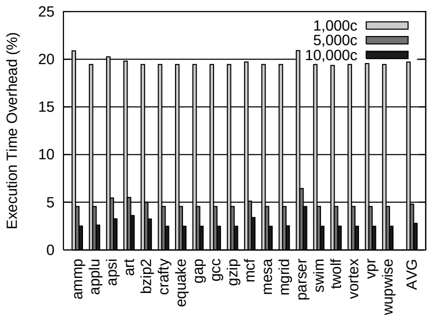

Figure 9. Execution time overhead.

speedup across our nine parallel benchmarks. Speedups are normalized to the performance of a single thread of execution on the baseline CMP. In addition, Figure 10 reports speedups for benchmarks with the largest overheads (barnes, and smallest overhead, kmeans). On average, when using a 64-entry age table, the performance overhead for up to 8 threads is between 4% and 5%. A sensitivity study on the number of age table entries shows less than a 4% reduction in execution time overhead when a 1024-entry age table is used (Section 4). Overall, these results suggest that modest hardware additions are adequate to minimize the performance overhead of DCC on parallel applications.

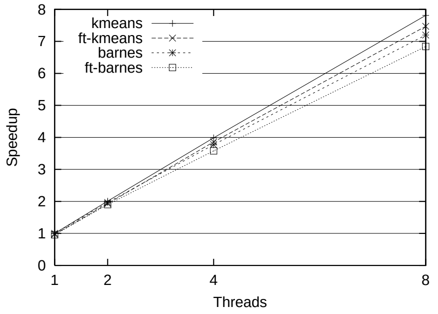

Figure 10. Speedup on baseline and fault tolerant CMPs (marked ft) for parallel benchmark with largest and smallest overheads. All curves are normalized to a sequential run on the baseline CMP.

5.1.2 Comparison Against Relaxed Input Replication

DCC utilizes long checkpoint intervals to amortize the cost of dynamic coupling. To maintain input coherence between redundant threads, we introduce an age table to track open read windows. Reunion [23] arguably proposes a conceptually simpler scheme of relaxed input replication: input incoherence may occur, but it would be detected as a fault. To guarantee forward progress, Reunion single-steps the cores to the first load instruction. Reunion still relies on dedicated communication channels for output comparison, and thus cannot provide the flexibility of dynamic coupling. Nevertheless, we would like to asses the performance of relaxed input replication under DCC's larger checkpoint intervals, and to quantitatively establish whether we need our age table mechanism.

Figure 11 shows the slowdown of Reunion's relaxed input scheme compared to DCC's age table scheme across our parallel benchmarks. For applications that have little read-write sharing, such as blast and swim, relaxed input replication incurs relatively modest overhead. However, applications that have more read-write sharing, raytrace and scalparc, incur significantly higher execution times (more than two-fold for 10,000-cycle intervals).

Relaxed input replication performs poorly in this context for two main reasons. First, as the checkpoint interval increases the redundant pair of cores becomes progressively out of synch. We have noticed differences of up to a few hundred instructions for the longer checkpoint intervals. This results in more opportunity for an intervening store to cause input incoherence. Second, single stepping the execution to the first load instruction does little to synchronize cores when the checkpoint interval may execute thousands of loads. If the offending memory operation occurs at the end of the interval, many rollbacks will ensue before single-stepping brings the synchronized execution close to that operation. Overall, the performance of relaxed input replication deteriorates quickly, and is inadequate for DCC's larger checkpoint intervals.

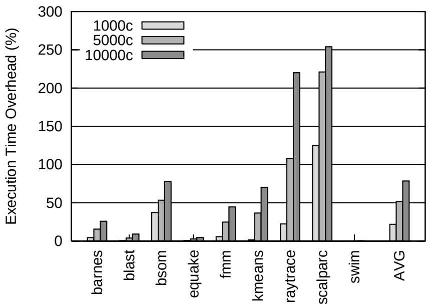

Figure 11. Slowdown of Reunion's [23] relaxed input replication with respect to DCC.

5.1.3 Performance under Manufacturing Defects

DCC degrades half as fast as mechanisms that rely on static DMR pairs when confronted with manufacturing defects, process variations, and wearout. While a defective or excessively leaky core renders both cores in a static DMR pair dysfunctional, DCC can utilize all functional cores regardless of their physical location or adjacency. Figure 12 compares DCC to an ideal, overhead-free static-DMR scheme on eight- and 16-core CMPs with two defective cores. The y-axis shows the speedups achieved by both schemes over a sequential run without fault tolerance. Reported speedups account for the small fraction of cases where two defective cores may fall into the same static DMR pair, in which case DCC does not offer an advantage. We account for such cases by generating 100K CMP configurations with two defective cores, where defect locations are sampled from a uniform random distribution. We report the average speedup over these 100K chips, which include chips with two failures in a single static-DMR pair.

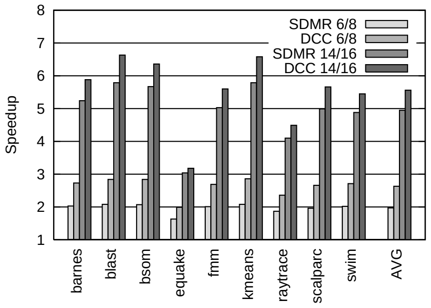

Figure 12. Average speedup of DCC and ideal static coupling on 8- and 16-core CMPs with two defective cores.

On an eight-core CMP, DCC achieves an average speedup of 2.63 across all applications, while static-DMR's speedup is only 1.97. For the sixteen-core CMP, DCC and static DMR obtain average speedups of 5.56 and 4.95, respectively. These results suggest that flexible DMR frameworks like DCC are an attractive to construct gracefully-degrading, fault-tolerant CMP designs that can meet deep submicron challenges.

	Threads	Speedup Of Benchmarks									
		barnes	blast	bsom	equake	fmm	kmeans	raytrace	scalparc	swim	average
Baseline	1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	2	1.97	1.99	2.00	1.62	1.96	2.01	1.83	1.89	1.95	1.91
	4	3.77	3.96	3.96	2.57	3.68	3.99	3.20	3.66	3.70	3.61
	8	7.20	7.93	7.66	3.66	6.62	7.81	5.38	6.53	6.33	6.57
DCC	1	0.96	0.97	0.96	0.97	0.96	0.97	0.97	0.98	0.96	0.97
	2	1.90	1.91	1.92	1.56	1.87	1.93	1.76	1.84	1.88	1.84
	4	3.58	3.84	3.82	2.48	3.52	3.86	3.06	3.51	3.57	3.47
	8	6.84	7.63	7.21	3.49	6.38	7.47	5.04	6.24	6.03	6.25

Table 5. Speedup of parallel applications with 1, 2, 4, and 8 threads for both the baseline CMP and DCC.

6 Conclusions

We have presented dynamic core coupling (DCC), an inexpensive DMR mechanism for CMPs, which allows arbitrary processor cores to verify each other's execution without requiring dedicated communication hardware. By avoiding static binding of cores at design time, DCC degrades half as fast in the presence of errors and can recover from permanent faults without the need for constant TMR or dedicated spares.

Our evaluation has shown the performance overhead of DCC to be 3% on SPEC2000 benchmarks, and within 5% for a set of scalable parallel scientific and data mining applications with up to eight threads (16 cores). We have also seen that DCC can offer significant performance improvements compared to static DMR schemes. Overall, we have shown that flexible DMR frameworks like DCC hold significant performance potential when confronted with the challenges of deep submicron process technologies in current and upcoming CMPs.

7 Acknowledgments

We thank Meyrem Kırman, Nevin Kırman, and the anonymous reviewers for useful feedback. This work was funded in part by NSF awards CCF-0429922, CNS-0509404, CAREER Award CCF-0545995, and an IBM Faculty Award (Martínez); by NSF awards CNS-0435190, CCF-0428427, CCF-0541321, and the DARPA/MARCO C2S2 Center (Manohar); and by equipment donations from Intel.

References

[1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, February 2005.

[2] Guido Albertengo and Riccardo Sisto. Parallel CRC generation. *IEEE Micro*, 10(5):63–71, 1990.

[3] Shekhar Borkar, Tanay Karnik, Siva Narendra, Jim Tschanz, Ali Keshavarzi, and Vivek De. Parameter variations and impact on circuits and microarchitecture. In *Design Automation Conf.*, June 2003.

[4] Shekhar Y. Borkar, Pradeep Dubey, Kevin C. Kahn, David J. Kuck, Hans Mulder, Stephen S. Pawlowski, and Justin R. Rattner. Platform 2015: Intel processor and platform evolution for the next decade. In *Technology@Intel Magazine*, March 2005.

[5] Cristian Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.

[6] Kypros Constantinides, Stephen Plaza, Jason Blome, Bin Zhang, Valeria Bertacco, Scott Mahlke, Todd Austin, and Michael Orshansky. Bulletproof: A defect-tolerant CMP switch architecture. In *Intl. Symp. on High Performance Computer Architecture*, February 2006.

[7] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *Intl. Symp. on Computer Architecture*, June 2003.

[8] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, 2000.

[9] Meyrem Kırman, Nevin Kırman, and José F. Martínez. Cherry-MP: Correctly integrating checkpointed early resource recycling in chip multiprocessors. In *Intl. Symp. on Microarchitecture*, December 2005.

[10] AJ KleinOsowski and David J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *IEEE Computer Architecture Letters*, 1(2), 2002.

[11] José F. Martínez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Intl. Symp. on Microarchitecture*, November 2002.

[12] Dennis McEvoy. The architecture of Tandem's NonStop system. In *ACM'81*, November 1981.

[13] Patrick J. Meaney, Scott B. Swaney, Pia N. Sanda, and Lisa Spainhower. IBM z990 soft error detection and recovery. *IEEE Trans. on Device and Materials Reliability*, 5(3):419–427, 2005.

[14] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Intl. Symp. on Computer Architecture*, May 2002.

[15] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Intl. Symp. on Microarchitecture*, December 2003.

[16] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[17] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Intl. Symp. on Computer Architecture*, June 2000.

[18] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, 2005. http://sesc.sourceforge.net.

[19] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Intl. Symp. on Fault-Tolerant Computing*, June 1999.

[20] L. Sherman. Stratus continuous processing technology – the smarter approach to uptime. Technical report, Stratus Technologies, 2003.

[21] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Intl. Conf. on Dependable Systems and Networks*, June 2002.

[22] T. J. Slegal, Timothy J. Slegel, Robert M. Averill III, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[23] Jared C. Smolens, Brian T. Gold, Babak Falsafi, and James C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Intl. Symp. on Microarchitecture*, December 2006.

[24] Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Fingerprinting: bounding soft-error detection latency and bandwidth. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[25] Ed Sperling. Turn down the heat…please, March 2007. http://www.edn.com.

[26] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The case for microarchitectural awareness of lifetime reliability. In *Intl. Symp. on Computer Architecture*, June 2004.

[27] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The impact of technology scaling on lifetime reliability. In *Intl. Conf. on Dependable Systems and Networks*, June 2004.

[28] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Intl. Symp. on Computer Architecture*, June 1995.

[29] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Intl. Symp. on Computer Architecture*, May 2002.